Army Research Laboratory

ARL

Android Video Streaming

by Jonathan Fletcher, David Doria, and David Bruno

ARL-TR-6947 May 2014

Approved for public release; distribution is unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer’s or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-6947 **May 2014**

Android Video Streaming

Jonathan Fletcher, David Doria, and David Bruno
Computational and Information Sciences Directorate, ARL

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE						*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**						
1. REPORT DATE *(DD-MM-YYYY)* May 2014		**2. REPORT TYPE** Final				**3. DATES COVERED (From - To)** July 2013–September 2013
4. TITLE AND SUBTITLE Android Video Streaming						**5a. CONTRACT NUMBER**
						5b. GRANT NUMBER
						5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Jonathan Fletcher, David Doria, and David Bruno						**5d. PROJECT NUMBER**
						5e. TASK NUMBER
						5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-CIH-S Aberdeen Proving Ground, MD 21005-5067						**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-6947
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)						**10. SPONSOR/MONITOR'S ACRONYM(S)**
						11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT Soldiers often bring their own mobile devices into the field, and those devices are untapped resources that can be used as cheap sensors. To use the sensors in this manner, significant data communication is required, and, unfortunately, bandwidth is a scarce resource on the battlefield. Establishing and using wireless ad hoc connections between devices for data transmission can help augment the wireless network to provide more bandwidth. Our goal in this work was to create a proof-of-concept application that demonstrates the feasibility of using Android devices in a wireless ad hoc network to transmit some of the information gathered by these sensors. This report will discuss a prototype application that enables streaming video from one Android device to another over a wireless ad hoc network established between devices that are in range of each other.						
15. SUBJECT TERMS Android, video, streaming, ad hoc						
16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**		**19a. NAME OF RESPONSIBLE PERSON** David Bruno
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	28		**19b. TELEPHONE NUMBER** (*Include area code*) 410-278-8929

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

INTENTIONALLY LEFT BLANK.

1. Introduction

The Army is interested in developing applications that take advantage of computing power, which is becoming more readily available on the battlefield. One such application is video collection and processing. Soldiers often bring their own mobile devices into the field, and those devices are untapped resources that can be used as cheap sensors. To use the sensors in this manner, significant data communication is required, and, unfortunately, bandwidth is a scarce resource on the battlefield. For this reason, we are interested in establishing and using wireless ad hoc connections between devices for data transmission. Our goal in this work was to create a proof-of-concept application that demonstrates the feasibility of using devices in this manner. Our prototype application enables streaming video from one Android device to another over a wireless ad hoc network established between devices that are in range of each other.

The video captured by the Soldiers' devices could have many possible uses. It could be used to provide a source of data that can be processed by a nearby high-performance computing asset and returned to a squad of Soldiers with annotations indicating the location of friendly and enemy entities, facial recognition results, or other relevant information. The video streams could also provide video chat capabilities or remote reconnaissance.

Using mobile devices as tools in the battlefield provides easily replaceable equipment to the Soldier at a low cost. Most existing military tools are developed by contractors and include their own proprietary hardware and software systems, making any type of universal development difficult. With the growing popularity of mobile devices and the development of the Android operating system (OS), targeting this platform was a natural choice.

In this document, we describe several aspects of video streaming and the challenges of performing video streaming between Android-based devices. In section 2, we discuss some of the fundamental concepts of video streaming. In section 3, we detail the requirements, design, and implementation of our video streaming application in Android. Finally, in section 4 we summarize the work and make recommendations for the development of future versions of this type of system.

2. Video Streaming

Video streaming is the process of viewing a video on one device that is delivered in a "just-in-time" fashion from another device on a network. There are two major components to video streaming: the video codec and the transmission protocol. These two components contain several parameters that can affect the quality of the video stream.

2.1 Video Encoders and Decoders

To encode something means to convert it from one format to another. For the video encoders, the conversion is from a sequence of images into a continuous stream of data that represent an initial image and the changes to the image over time. The encoded format for video is either *lossy* or *lossless*. Lossy encoding means that the images used to represent the original sequence of images are not exact representations of the original sequence of images but are instead an approximation of those images. Conversely, lossless encoding means that the encoded data can exactly represent the original information. A video encoder is a piece of software or hardware that performs the encoding of the sequence of images into a desired format. Video encoding is typically lossy so that the amount of memory required to store it is reduced.

To decode a video means to convert it from its current encoded state back to the original data. When a lossy encoder is used to convert video, it is no longer possible to retrieve the exact sequence of images given.

Video encoders and decoders have existed for quite some time and are usually combined into a single package termed a *codec*. Some of the most commonly known video codecs include MPEG-1, MPEG-2, MPEG-4, H.263, H.264, VP8, and MJPEG. Implementations of these codecs are readily available for desktop platforms. One problem with these video codecs is that they require significant computation to encode and decode video files. On an Android device, that computational power also means using a significant amount of energy which can quickly drain the battery. To compensate for this, Android devices incorporate specialized hardware to perform the encoding and decoding. Unfortunately, this means that only a select few of the most common codecs are available on all of the Android devices.

2.2 Transmission Protocols

Typically we refer to the device that is recording the data as the *server* and the device that is receiving the video stream as the *client*. Once the video data has been encoded on the server it must be made available to the client. Many protocols exist for transferring data from a server to a client, but only a few of them are capable of "streaming" data (as opposed to simply copying a file in its entirety with no latency constraints).

2.2.1 HyperText Transfer Protocol (HTTP)

As its name suggests, the HyperText Transfer Protocol (HTTP) was originally designed to transfer files containing formatted text written in the HyperText Markup Language (HTML) across a network so that it can be displayed on a remote machine. In order for the text to be read properly, it is important for every piece of the text to be transferred to the remote machine correctly. HTTP is designed to transmit using the Transmission Control Protocol (TCP), which is the component responsible for the ensuring the successful delivery of data packets. If a packet is dropped or missed, a client that is supposed to receive the TCP packet will send a request back to the server to resend the packet.

While the original intent of HTTP was to transfer these HTML files, over time it has been modified to allow the transfer of many different types of data, including video. This method of transferring video has been termed *progressive download* because of the requirement of HTTP to transfer files. This is the typical method used when a stream is produced from a file on a remote server. Data can be transferred in chunks reconstructed on the client. This type of method is used in consumer Digital Video Recorders (DVRs) or when viewing prerecorded video clips on the Internet. Once a video file transfer has been started, video player applications (the client side of the video stream) can start displaying the video on a screen up until it reaches a portion that has not yet been received.

One of the problems with using this method for live streaming occurs when packets are dropped. When this happens, the TCP layer will request the packets to be retransmitted. The video player will then pause while those packets are again sent by the server. This pausing will cause a gradually increasing delay in the video. A method to counter this pausing is to buffer the data sufficiently so that out-of-order retrieval of the file is inconsequential. Unfortunately, using a buffer causes issues for live streaming. When using a buffer, the client is playing video data that has been received several seconds after the time that the video was sent, which causes an apparent delay in the video. For live broadcast situations, such as sports events or news broadcasts, this small delay is not an issue. However, for a live transmission that requires interaction, the delay is troublesome.

Another problem that can occur, especially in an ad hoc network, is insufficient bandwidth to transfer the data in real time. Insufficient bandwidth will cause the video player to constantly run out of already-transferred video data, which will cause the display to be jittery. A potential solution to this problem is once again to use a buffer on the client, but again, this buffer can introduce the same problems with delay as in the live transmission case. Another solution is to change the resolution, bitrate, and/or framerate of the video being transmitted to the client, reducing the bandwidth requirements of the video. This solution is typically not viable because a progressive download is required to have a constant resolution, bitrate, and framerate because the data are eventually interpreted as a single file. Using HTTP in a progressive download manner allows only these parameters to be modified when the stream is stopped and then restarted.

A final problem with progressive download is related to the file containers. Raw encoded video data are nothing more than a sequence of frames. The timing of the sequence of frames is left to the file container to describe. This is done because there is often audio data that accompany the video data, and the audio needs to be synchronized with the video. The container file formats describe the timing of the frames. Video playback can start only once that section of the container describing the frame timing has been received. The way that progressive download works is by placing that timing section at the beginning of the file. If that section is not placed at the beginning, then the entire file must be downloaded before playback can begin because HTTP does not have a mechanism to send timing information about the video to the client. In a live video stream, the full video does not yet exist so the timing cannot be known, causing the playback of the video to wait until the server has finished recording and transmitting the timing section.

More useful information about HTTP streaming can be found in Siglin (*1*).

2.2.2 Real-Time Transport Protocol (RTP) and RTP Control Protocol (RTCP)

In a live video stream, it is very important to minimize the delay of the video. It is often advantageous to simply ignore dropped packets and try to recover from the loss of data while keeping the video playing. For the network to behave in this manner, the video data cannot be transported using TCP. For this behavior, the Real-Time Transport Protocol (RTP) (*2*) was devised to use a lower layer of the networking stack that does not rely upon packet resending. By using the User Datagram Protocol (UDP), the client will not examine the packets to determine if all of them have been received, and we avoid the potential delay introduced by resending packets.

RTP was created as a generic method to transfer raw data of any kind from a server to a client in real time. For video data, this means that the timing information does not exist for the video frames encapsulated in the RTP packets. To assist with the control of the data that are transferred by RTP, a sister protocol called RTP Control Protocol (RTCP) (*3*) was designed to help track quality statistics and describe how to use the data.

A detailed specification for RTP and RTCP can be found in the Request For Comments (RFC) 3550 (*4*). The RFC 3550 document describes a generic method to transfer data over RTP, but different types of data may have different requirements regarding how the data are turned into individual packets to be transmitted. In some types of data, there can be minor differences that the generic specification does not cover. Individual RFC documents have been written to specify how particular types of data are to be transmitted. These special types include, but are not limited to, H.263 video (specified in RFC 4629 [*5*]), H.264 video (specified in RFC 6184 [*6*]), AMR audio (specified in RFC 3267 [*7*]), and AAC audio (specified in RFC 3640 [*8*]).

2.2.3 Real-Time Streaming Protocol (RTSP)

It is possible to use RTP by itself provided that there is a way to describe the RTP data that are sent. One way to do this is to use the Session Description Protocol (SDP). SDP is used to initiate the stream, allowing RTP to continue. Unfortunately, the Android OS does not have support for SDP directly. The only Android-supported way to transfer the SDP information is via the Real-Time Streaming Protocol (RTSP). RTSP is a protocol designed to control how the user interacts with an underlying stream. RTSP is a state machine describing the current state of the stream. There are 10 distinct states: DESCRIBE, ANNOUNCE, GET_PARAMETER, OPTIONS, RECORD, REDIRECT, SET_PARAMETER, SETUP, PLAY, and PAUSE. When a client sends a DESCRIBE message to the server, the server will respond with the SDP information, which allows the client to know all the details about the RTP stream. Next, the client will issue a

SETUP command, which will open the sockets on both ends readying the devices to send and receive the RTP and RTCP information. Finally, the client will issue a PLAY message, and the server will start sending the RTP data and RTCP messages across the sockets to the client. A detailed specification of RTSP can be found in RFC 2326 (*9*).

3. Implementation of Video Streaming in Android

3.1 Requirements

In this section, we discuss the design and implementation of a proof-of-concept system, an Android application called ARL Video Streaming, which allows video streaming between Android devices for use on the battlefield. This environment requires U.S. Army Research Laboratory (ARL) researchers to carefully consider many aspects of the design. When on the battlefield, Soldiers have limited access to a power source to charge the battery of their mobile device. Because of this, it is important to keep energy efficiency in mind while designing the streaming application.

Another interesting consideration is that a bright screen can alert enemy combatants to the location of a Soldier. To prevent this, streaming needs to continue even while the device's screen is turned off.

A "one-to-many" stream, where several clients can view the video from a single server simultaneously, is a desirable property of a streaming system. By simply treating all clients individually, the server device can quickly run out of bandwidth, as the stream must be sent once for each connected client. A potential solution to this problem is to use multicast network communication, which only requires one stream to be sent from the server while allowing all of the clients to receive the stream. This approach is discussed in section 3.5.3.

3.2 Media Formats and Network Protocols

There are specific media formats and network protocols that are supported by Android. According to the Android Developer's Web site (*10*), Android's MediaRecorder object can only work with H.263, H.264, and MPEG-4 SP codecs. Furthermore, the Android Developer's Web site (*11*) also specifies that streaming is only supported through HTTP and RTSP. Upon further investigation, the "supported" streaming methods only implement receivers, not transmitters. Android does not have any support for sending a video stream from an Android device to a remote client.

To provide an Android-based server for the video streaming, we built on top of tools from an open-source library called Spydroid (*12*). Spydroid provides an H.263 and H.264 codecs for use with both RTSP and HTTP protocols.

To validate which media format and network protocol combinations worked, a reliable video client application was needed. Typically something like VideoLAN Client (VLC) is used for this purpose in a desktop environment. However, while VLC is a very mature application on Windows and Linux, VLC for Android is still in a beta testing phase, and versions have only been developed to work with a small set of devices. Another client, MX Player (*13*), enabled us to test specific combinations of codecs and protocols.

After finishing the initial testing, we found that only some of the combinations would work. Since there were at least a few combinations that were successful, a simple client was implemented in Android that used the Android built-in MediaPlayer class. A URL is all that is needed as input for the MediaPlayer class, and the MediaPlayer will request and play the video stream. A summary of the compatibility of the different applications with each combination of codecs and protocols is shown in table 1.

Table 1. Video streaming results with MX Player and a simple client.

Codec	**RTSP**	**HTTP**
H.263, MX Player	Failed	Success
H.264, MX Player	Success	Failed
H.263, simple client	Failed	Failed
H.264, simple client	Success	Failed

After running this test, we found that only one combination of protocol and codec was able to successfully transmit a video stream. The codec that was successful across all tests was H.264, which, fortunately, is one of the hardware-supported codecs.

3.3 Software Code Design

3.3.1 Structure of the Client Code

Most of the client portion of the ARL Video Streaming application is handled by an Android built-in class called MediaPlayer. The MediaPlayer is passed a string, which is the URL of an RTSP-based stream. After receiving the URL, the MediaPlayer handles all the steps necessary to set up, receive, and play the video stream.

The MediaPlayer object also requires a View to display the video stream on the device’s screen. A TextureView was used because it has the ability to rotate the video being displayed. The rotation could potentially allow the software to rotate the video so that the stream’s “bottom” would always be shown at whichever part of the screen was closest to the ground.

3.3.2 Structure of the Server Code

The ARL Video Streaming application server code builds off of Spydroid, an open-source Android app that acts as a video and audio server. The structure of the program is shown in figure 1.

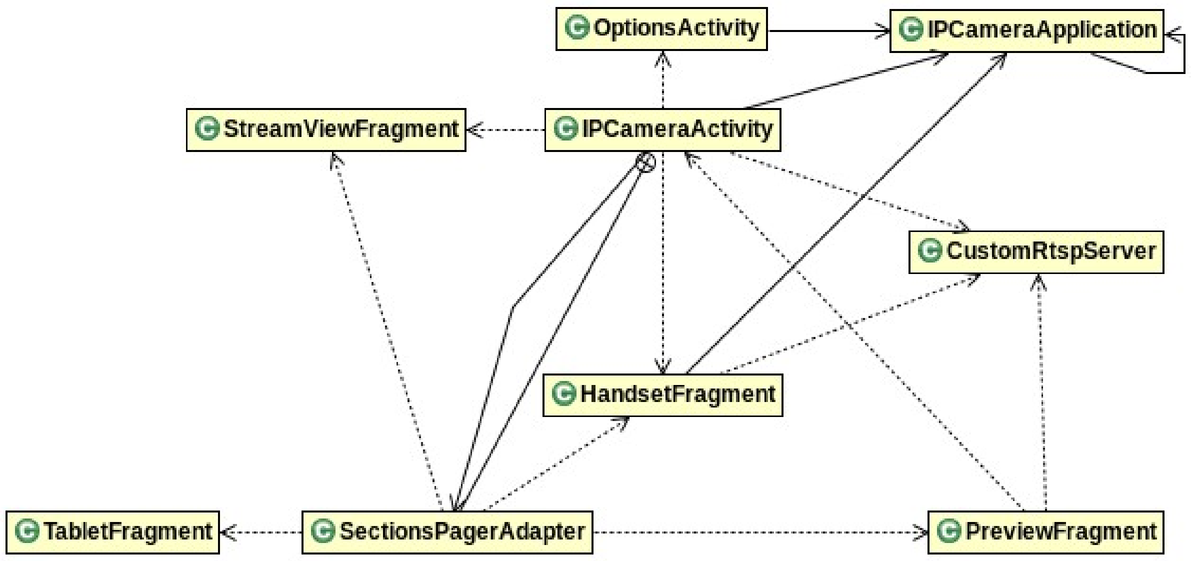


Figure 1. Video Streaming graphical user interface (GUI) class interaction.

All Android applications start with a single instance of the Application class. Inside of this Application, there are several instances of the Activity class of which only one can be active at a time. The Activities are declared in the "AndroidManifest.xml" file. An Activity is always associated with a view, which means the Activity always has a GUI component associated with it.

In the ARL Video Streaming application, the instance of the Android Application class is called IPCameraApplication, as shown in figure 1. There are two Activity instances in the GUI portion of ARL Video Streaming. IPCameraActivity is the main Activity that is first instantiated when the users start the application. Inside of this class is a PageViewer instantiation, which is used to switch between the different tabs. Once a video stream is started, it will continue streaming as long as this Activity is active. Once the user switches to a different Activity, the streaming Server will shut down and streaming will stop. OptionsActivity is the secondary Activity. It is accessed through a menu item and allows the user to select from the available protocols, bitrates, and server type to transmit. When the OptionsActivity is activated, then the IPCameraActivity will be placed into a "Paused" state or a "Stopped" state (these states are handled by the Android OS), and the streaming will stop.

The IPCameraActivity class contains the Android built-in class ViewPager, which facilitates swiping through pages. An Android built-in class called PagerAdapter is used to connect the ViewPager and the list of pages, which are called Fragments. FragmentPagerAdapter, which extends the PagerAdapter class, holds several Fragments that contain displays of information about how to connect to the device, the camera preview, and the video transmitted from a remote device.

The CustomRtspServer class is the object that performs the communication via the RTSP protocol. This main object is instantiated in each of the Fragments inside of the IPCameraActivity. This instance is configured to be a “service,” which means that it can perform other operations separately from the main thread of the application. In this case, the operations we wish the Service to perform are receipt and response of network socket communication. CustomRtspServer will wait in an idle state, listening on a socket for a request of a stream to be started. Once the request is received, a Session is created by way of the SessionManager. Figure 2 illustrates how the CustomRtspServer uses the SessionBuilder to create the instance of an H264Stream, which is ultimately used by a Session. After the CustomRtspServer instantiates a new Session, control of that Session is relinquished, and the CustomRtspServer will once again wait listening on the socket for another request. In this manner, a Session will also be a part of the service so that the transmission of the data could continue in the background.

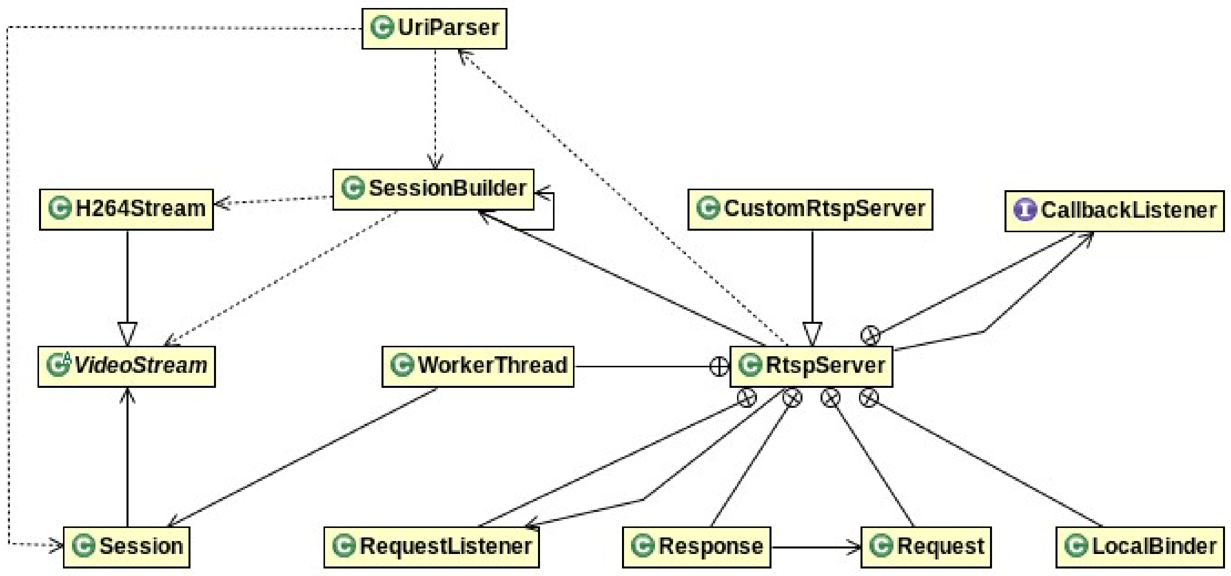


Figure 2. RTSP Server class interaction.

A Session can be thought of as an instance of a stream. As shown in figure 3, the Session contains all of the information about the stream and will contain instances of the Camera, MediaRecorder, and VideoStream. The Android built-in Camera class is responsible for interacting with the physical camera in the device and converting the optical sensor data into a digital image. The Android built-in MediaRecorder class is responsible for capturing raw media and encoding it in a specified format.

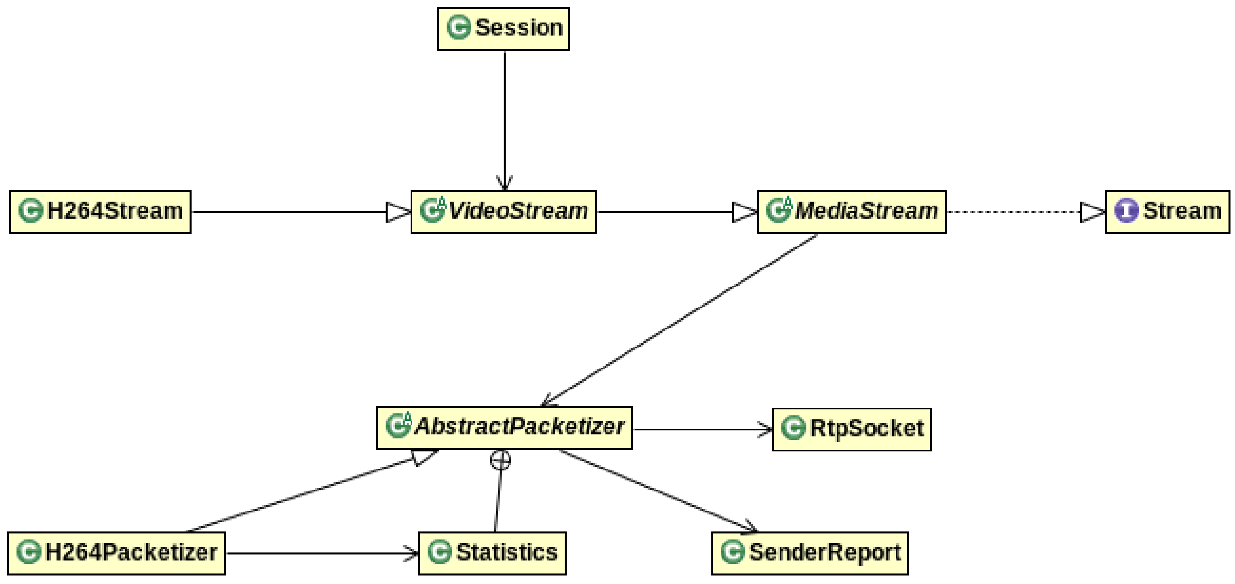


Figure 3. RTSP stream class interaction.

When video is desired by the application, an instance of the Camera class is passed to the MediaRecorder. The media encoding type, an output file, and a file type are required parameters to create a MediaRecorder. The Android MediaRecorder class was designed to save the video to a file. Since the goal of our software is to transfer the data out over a network and not to save the video to a file, we had to implement a way to bypass the requirement to save the video to a file. We did this by providing the MediaRecorder a FileDescriptor instance that originated from a LocalSocket object. The video data would be continuously written to the LocalSocket, which would be connected to another class internal to the video streaming application.

An AbstractPacketizer class is responsible for taking information from an InputStream (the other end of the LocalSocket instance coming from the MediaRecorder) and converting it to packets that are to be sent out over the network. Individual instances of each media type need to extend the AbstractPacketizer class.

The H264Packetizer is an extension of the AbstractPacketizer and implements the appropriate RFC for the given codec used to capture the video. After the H264Packetizer has created a packet, the packet is sent out over an RTPSocket instance, which contains a Java built-in MulticastSocket and DatagramPacket that will send the data across the physical network connection to the client.

3.3.3 Power Requirements

Our application has two distinct parts that have different power requirements: the server side, which sends the video stream, and the client side, which receives the video stream.

The server portion of the code is continuously running in the background, regardless of the state of the client portion of the application. The server will either be actively capturing a video and sending it out over the network, or it will be listening on a socket waiting for a request for a video stream. This means that the server needs the CPU to always be active.

The client portion of the software does not need the CPU to be considered. If the stream is active, then the CPU will be used. If the stream is not active, then the CPU will not be used. The client has a requirement to allow the user to disable the screen while still maintaining a video stream.

At first glance, it appeared that all of this functionality could be obtained by using a tool in Android's power manager called a wake lock. There are several kinds of wake locks. Based on the information from the Android Developer's reference site for the PowerManager (*14*) displayed in table 2, the "full" wake lock will allow both the CPU and screen to remain active. However, most of the wake locks, including the full wake lock, will allow the device to go into a sleep state when the power button is pressed. What this means is that pressing the power button to turn the screen off would cause the full wake lock to be released, and the application would eventually move into a paused or stopped state, which shuts down the video stream. The only wake lock that remains active once the power button is pressed is the "partial" wake lock. The functionality of the partial wake lock is to keep the CPU active and not attempt to modify the screen behavior from the system's default.

Table 2. Android PowerManager wake lock settings.

Flag Value	**CPU**	**Screen**	**Keyboard**
PARTIAL_WAKE_LOCK	On[a]	Off	Off
SCREEN_DIM_WAKE_LOCK	On	Dim	Off
SCREEN_BRIGHT_WAKE_LOCK	On	Bright	Off
FULL_WAKE_LOCK	On	Bright	Bright

[a]If you hold a partial wake lock, the CPU will continue to run, regardless of any display timeouts or the state of the screen, and even after the user presses the power button. In all other wake locks, the CPU will run, but the user can still put the device to sleep using the power button.

With the use of the partial wake lock, the CPU is always active, but the screen still does not follow the requirements outlined. The screen needs to stay on when viewing a stream and turn off only when the user presses the power button. Furthermore, if the user presses the power button to bring the screen back on, then the screen should come back on and stay on. This feature is achieved by using a flag tied to a particular window. By calling getWindow().addFlags (WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON) in the main activity, the activity will always keep the screen on, provided the activity is being viewed. Since it is required that the application remain active to maintain the GUI and subsequently maintain the camera feed, the screen will stay on.

3.3.4 Maintaining Access to the Camera

The Android OS requires the use of a Preview with the Camera by way of a Surface object. The typical programming models for Android attempt to destroy any GUI element not being viewed currently to keep as much memory free as possible. The result of the destruction of the GUI elements means that the Camera no longer has an instance of a Surface, and therefore the camera stops recording. A way to work around this requirement for a GUI element tied to the camera is to create an instance of that GUI element inside of a Service instead of in the main GUI. For this to work, the Surface must be created programmatically rather than at design time using the Android XML GUI layout tool. If the Surface exists inside of the Service, even when the GUI is destroyed, the Surface tied to the camera still exists inside of the Service and thus the camera can continue recording.

3.4 Software Functionality

The main GUI for the ARL Video Streaming application consists of two pages that can be swiped back and forth. A third page contains options that control the parameters of the video stream and can be accessed through the menu.

When the application is first opened, the user is presented with the main page, which provides information about how to connect to a device and how to connect to other devices. The top section is titled “View a remote stream.” This section provides a text edit field where the user can enter the URL of a remote device when initiating a video stream. The lower section titled “For others to view your stream” provides the URL that a remote user would need to connect to that device, as shown in figure 4.

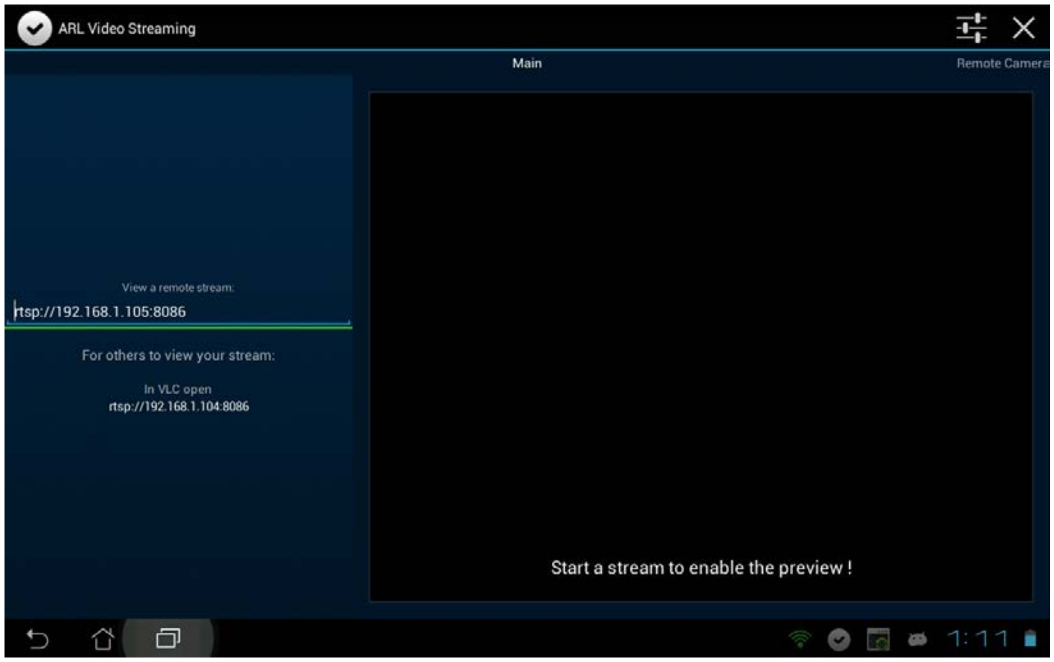

ARL Video Streaming
Main
Remote Camera
View a remote stream:
rtsp://192.168.1.105:8086
For others to view your stream:
In VLC open
rtsp://192.168.1.104:8086
Start a stream to enable the preview !
1:11

Figure 4. Main page while Wi-Fi is enabled, not streaming video.

This section also will display an indication that a connection is needed, as shown in figure 5. The preview of the device's camera is found on the right side of the main page. This preview helps the user controlling the server device to view what is being transmitted to the client device. Figure 6 shows the main page when on a tablet while sending a video stream to a client device.

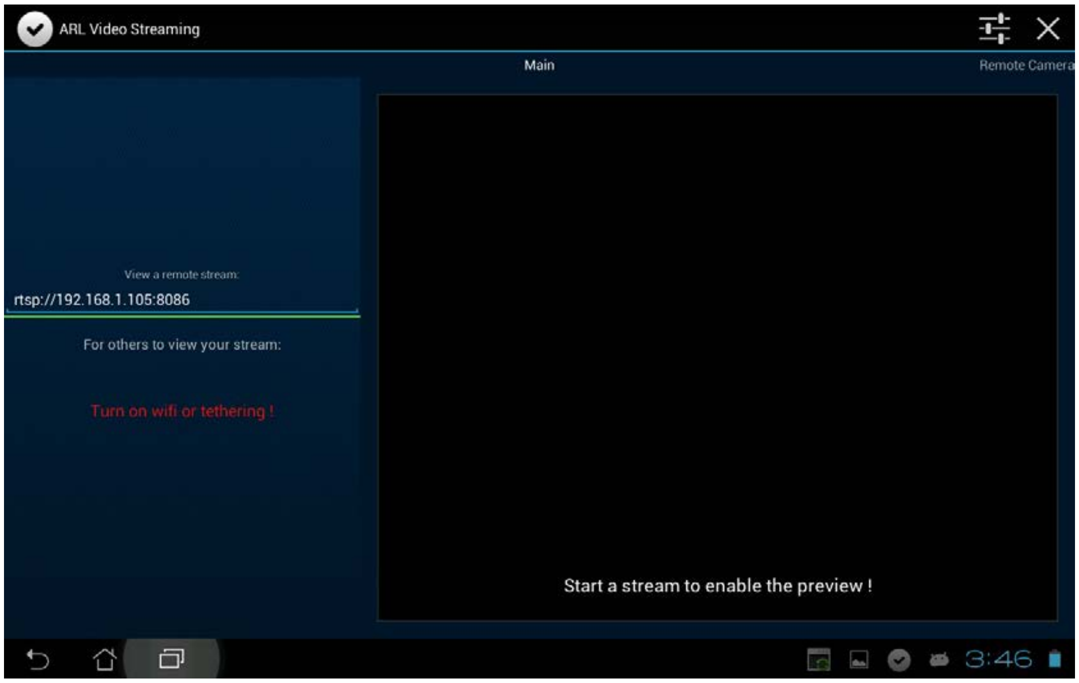

ARL Video Streaming
Main
Remote Camera
View a remote stream:
rtsp://192.168.1.105:8086
For others to view your stream:
Turn on wifi or tethering !
Start a stream to enable the preview !
3:46

Figure 5. Main page while Wi-Fi is disabled.

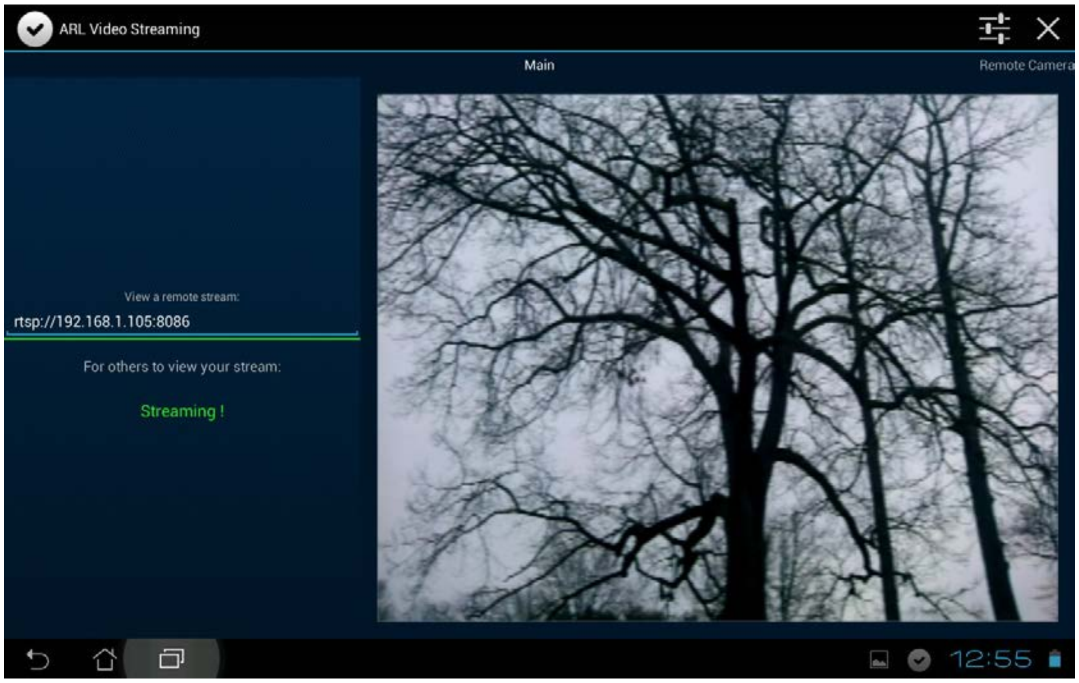


Figure 6. Main page while Wi-Fi is enabled, streaming video.

Swiping to the left from the main page accesses the second page labeled “Remote Camera” (shown in figure 7), a display of the remote camera’s video stream is visible. Two buttons to “Connect” and “Disconnect” can be found in the upper-left corner. Next to these buttons, a timer display was placed in the app to allow timing tests of the video stream.

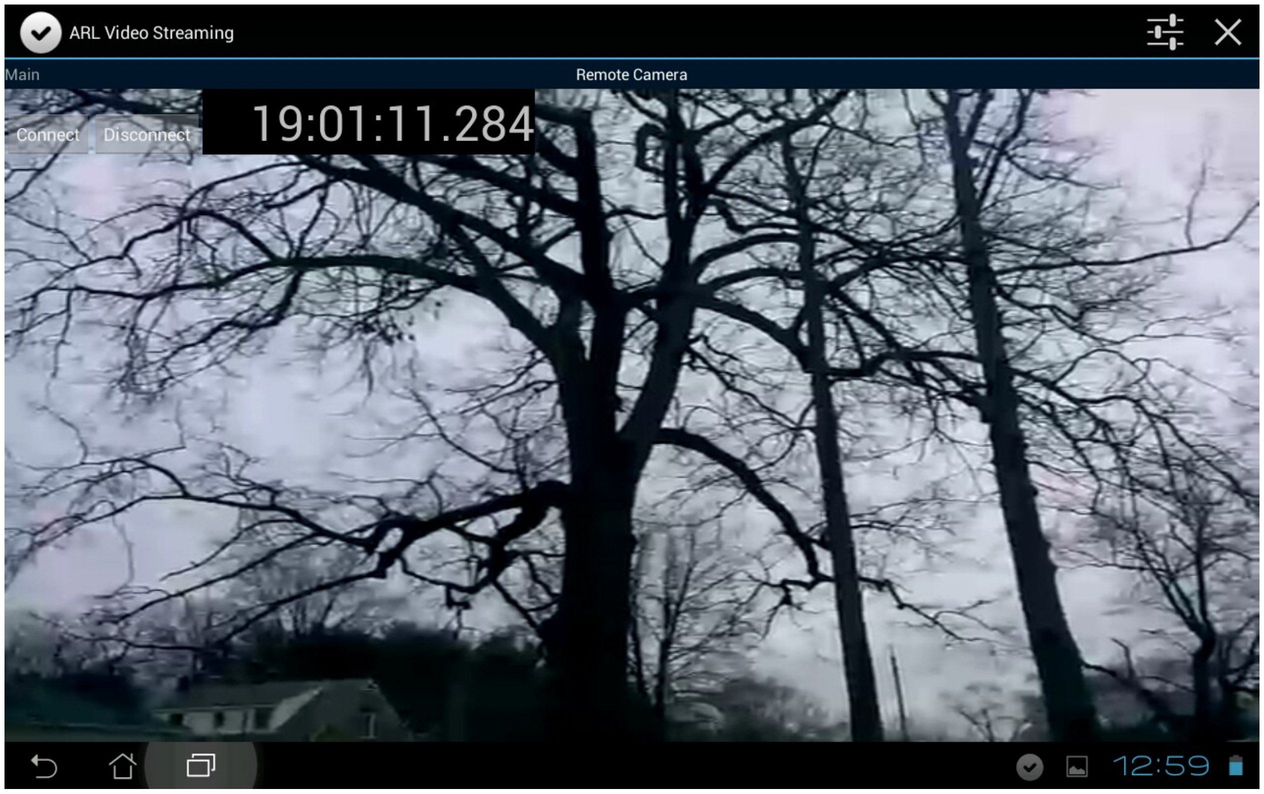
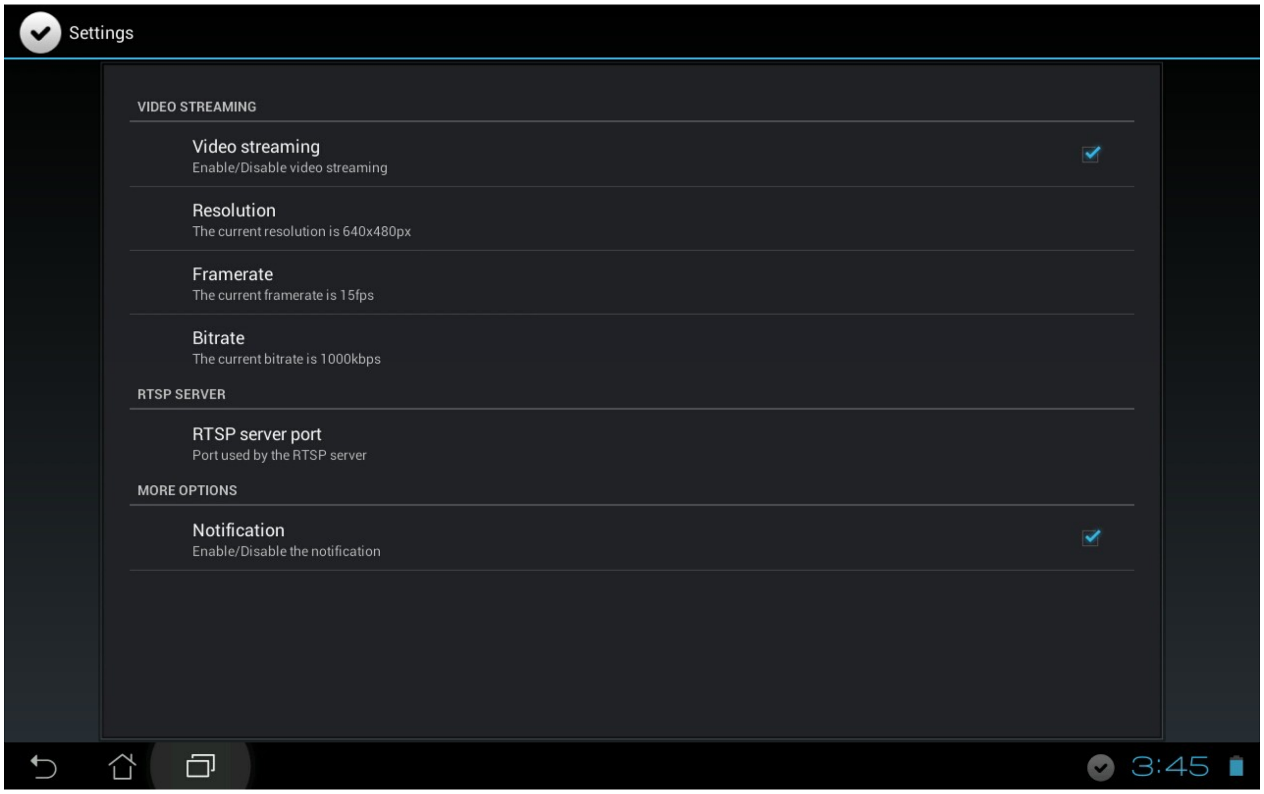


Figure 7. Remote Camera page while streaming.

The third page labeled “Settings,” shown in figure 8, can be accessed by pressing the menu item in the upper-right corner. This page contains three sections: Video Streaming, RTSP Server, and More Options.


Settings

VIDEO STREAMING

Video streaming ✓
Enable/Disable video streaming

Resolution
The current resolution is 640x480px

Framerate
The current framerate is 15fps

Bitrate
The current bitrate is 1000kbps

RTSP SERVER

RTSP server port
Port used by the RTSP server

MORE OPTIONS

Notification ✓
Enable/Disable the notification

3:45

Figure 8. Settings page.

The Video Streaming section contains information relevant to the codec for the video stream: Enable/Disable Video Streaming, Resolution, Framerate, and Bitrate. The RTSP Server section contains a setting to adjust the Port to use to listen for requests to start an RTSP stream. The More Options section contains a setting for displaying the icon in the notification bar.

3.5 Testing Results

3.5.1 Ad Hoc Network

Because of the desire to perform streaming over a mobile ad hoc network, a custom Android kernel was needed that has modified the wireless network driver to allow the device to enter an ad hoc mode. The details on the process of making the device operate in ad hoc mode can be found in the ARL technical report *Mobile Ad-Hoc Networking on Android Devices* (*15*).

We made an attempt to use a custom ROM, prepackaged by a third party, which had the necessary change in the wireless network driver. Unfortunately, there were changes to that ROM that also negatively affected the video streaming capabilities. Taking the original stock ROM and modifying it as described in *Mobile Ad-Hoc Networking on Android Devices* (*15*) allowed the video streaming to work properly over an ad hoc network.

3.5.2 Streaming Results

When using the H.264 codec, we noticed a delay between when the video is displayed in the preview screen on the server device and when the video is finally decoded and displayed on the client device. To determine this delay, we added a timer to the client's display of the remote server's video stream. Next, the server's camera was pointed at the client's screen. The client would then have the original timer displaying on the screen and a second timer displaying on the screen that was transmitted over the video stream. A screen capture was taken of the client's screen to freeze both timers in a single frame.

The delay was then computed by subtracting the two times. As can be seen in figure 9, the delay is approximately 7 s. In the image, the timer at the top is the original timer on the client device, and the timer below is the view of the timer coming through the video stream. The image of the timer in the video stream is fuzzy because the application did not have the camera's auto-focus functionality implemented.

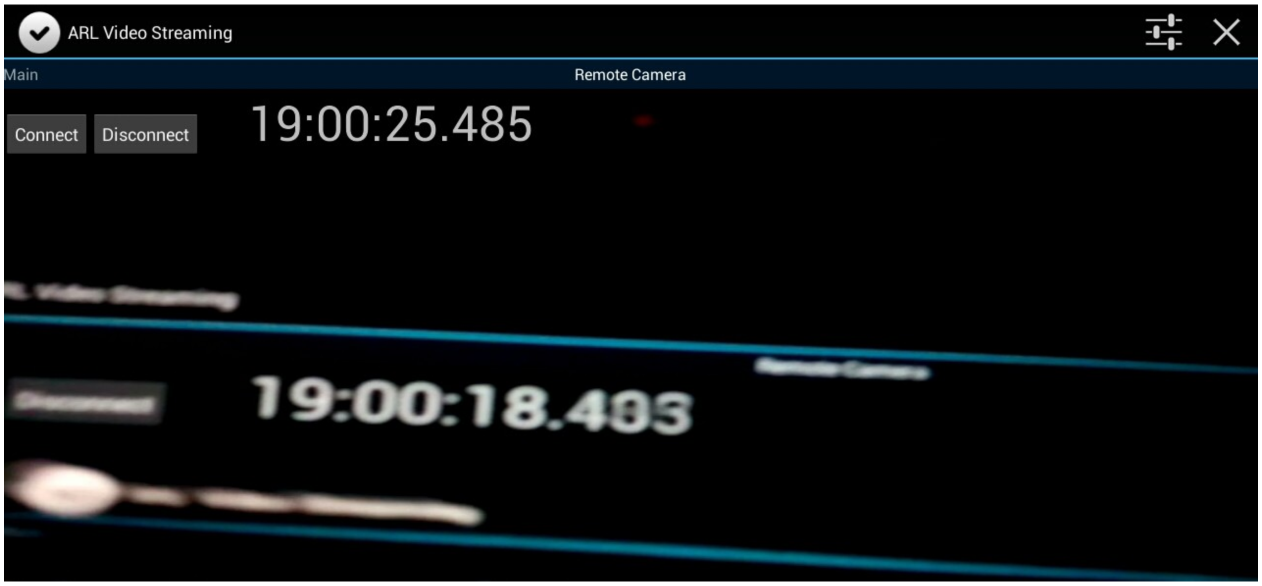


Figure 9. Video streaming delay.

3.5.3 Multicast Communication

To enable multiple clients to view a stream simultaneously, multicast communication can be employed. Before trying to use the MulticastSocket instance, one must inform the Android OS that this software wants permission to certain features by adding several permission flags to the AndroidManifest.xml file, which is a required file for every Android application (*16*).

```
<uses-permission
android:name=”android.permission.CHANGE_WIFI_MULTICAST_STATE” />
<uses-permission android:name=”android.permission.INTERNET”/>
<uses-permission android:name=”android.permission. ACCESS_NETWORK_STATE” />
<uses-permission android:name=”android.permission.ACCESS_WIFI_STATE”/>
```

Once the permissions are set properly, a MulticastLock is needed to allow access to send and receive multicast network communication.

```
WifiManager   wifi  = ( WifiManager )  mContext
                             .getSystemService( Context.WIFI_SERVICE ) ;
i f   ( wifi  != null )
{
     mLock = wifi
                    .createMulticastLock( “mil.army.arl.streaming” ) ;
     mLock.acquire( ) ;
}
```

After implementing UDP multicast streaming, we were unable to connect to the multicast stream from any client devices. Unfortunately, there are several issues in both the Android software and hardware that prevent this technique from working properly. Some Android manufacturers have made the network stack drop packets that are meant to be transmitted on a multicast socket (*17*). Eliminating these packets helps with conserving the battery life of the Android devices. Another issue is that some devices do not have an implementation of the Internet Group Manager Protocol (IGMP) (*18*). IGMP is meant to allow devices to know that someone is listening for multicast information so that the routing device can pass that information along. Some client Android devices never send out a message through IGMP that multicast information is desired, so the routing devices never transmit the packets destined for the multicast socket. There are several other known bugs listed on the Android Open Source Issue Tracker regarding multicast support on Android devices (*19–21*).

4. Conclusion

In this report, we have detailed some of the basic concepts of video streaming. We have investigated the capabilities that currently exist for video streaming from an Android device. We have also detailed requirements for video streaming on Android devices specific to a Soldier’s needs, developed an Android application based on these requirements, and tested the application. Performing video streaming with Android devices over an ad hoc network is feasible but not for all the scenarios originally discussed. The 7-s delay present in the video stream will make real-time use of the video stream infeasible, if not completely useless. The lack of multicast communication over the network means that the bandwidth of a server will be quickly saturated if multiple clients attempt to connect to that server.

If video streaming were ever required in the battlefield, significant work would be required to implement multicast of the ad hoc network. Also, the delay present in the video would need to be solved. These issues represent a roadblock of implementing video streaming on off-the-shelf Android devices. The current state of the video streaming application could be used to collect data for postprocessing later on, but real-time exercises and streaming would be difficult to implement.

Future work will focus on identifying the cause and reducing the delay in live video streaming. If the cause of the delay is identified it may be possible to reduce the time involved in displaying a live video stream across the established ad hoc network. Also, we will focus on implementing multicast capability, which would provide the ability to stream live to multiple clients. As of now a direct connection is necessary to receive the video stream. Multicast would allow for numerous clients of the video stream without directed connections.

The Android OS and its ecosystem of devices provide for a very powerful tool that can be used to incorporate off-the-shelf devices on the battlefield. Utilizing Android devices already owned by deployed Soldiers is a cost-saving method for generating more situational awareness in the battlefield. A large factor in making this possible includes creating the necessary capabilities with off-the-shelf hardware and unmodified software, such as Android running the devices. This project demonstrates that it is possible with existing commercial software (Android) and hardware to implement important capabilities that can be used on the battlefield. But as this work shows, more development is necessary to fully extend this capability to the battlefield with off-the-shelf hardware and software incorporated in most Android devices.

5. References

1. Siglin, T. HTTP Streaming: What You Need To Know. http://www.streamingmedia.com /Articles/Editorial/Featured-Articles/HTTP-Streaming-What-You-Need-to-Know -65749.aspx (accessed 30 September 2013).
2. Wikipedia. Real-Time Transport Protocol. http://En.Wikipedia.Org/Wiki/Real -Time_Transport_Protocol (accessed 30 September 2013).
3. Wikipedia. RTP Control Protocol. http://en.wikipedia.org/wiki/RTP_Control_Protocol (accessed 30 September 2013).
4. Schulzrinne, H.; Casner, S.; Frederick, R.; Jacobson, V. RTP: A Transport Protocol for Real-Time Applications, 2003. http://tools.ietf.org/html/rfc3550 (accessed 30 September 2013).
5. Ott, J.; Bormann, C.; Sullivan, G.; Wenger, S.; Even, R. RTP Payload Format for ITU-T Rec. H.263 Video, 2007. http://tools.ietf.org/html/rfc4629 (accessed 30 September 2013).
6. Wang, Y. K.; Even, R.; Kristensen, T.; Jesup, R. RTP Payload Format for H.264 Video, 2011. http://tools.ietf.org/html/rfc6184 (accessed 30 September 2013).
7. Sjoberg, J.; Westerlund, M.; Lakaniemi, A.; Xie, Q. Real-Time Transport Protocol (RTP) Payload Format and File Storage Format for the Adaptive Multirate (Amr) and Adaptive Multi-Rate Wideband (Amr-Wb) Audio Codecs, 2002. http://tools.ietf.org/html/rfc3267 (accessed 30 September 2013).
8. van der Meer, J.; Mackie, D.; Swaminathan, V.; Singer, D.; Gentric, P. RTP Payload Format for Transport of Mpeg-4 Elementary Streams, 2003. http://tools.ietf.org/html/rfc3640 (accessed 30 September 2013).
9. Schulzrinne, H.; Rao, A.; Lanphier, R. Real-Time Streaming Protocol, 1998. http://tools.ietf.org/html/rfc2326 (accessed 30 September 2013).
10. Mediarecorder.videoencoder. http://developer.android.com/reference/android/media /MediaRecorder.VideoEncoder.html (accessed 30 September 2013).
11. Supported Media Formats. http://developer.android.com/guide/appendix/media-formats.html (accessed 30 September 2013).
12. Spydroid-ipcamera. https://code.google.com/p/spydroid-ipcamera/ (accessed 30 September 2013).
13. Mx Player. https://sites.google.com/site/mxvpen/ (accessed 30 September 2013).

14. Android Developers Reference: PowerManager. http://developer.android.com/reference /android/os/PowerManager.html (accessed 30 September 2013).

15. Doria, D.; Fletcher, J.; Sookoor, T.; Jovel, F.; Bruno, D. *Mobile Ad-Hoc Networking on Android Devices;* ARL-TR-6845; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, 2014.

16. Android Developers Reference: MulticastSocket. http://developer.android.com/reference /java/net/MulticastSocket.html (accessed 30 September 2013).

17. UDP Multicast on Android. http://codeisland.org/2012/udp-multicast-on-android/ (accessed 30 September 2013).

18. Multicast and Android a Big Headache. http://www.programmingmobile.com/2012/01 /multicast-and-android-big-headache.html (accessed 30 September 2013).

19. Android Open Source Project - Issue Tracker, Issue #51195: Many Devices Have Multicast Disabled in the Kernel. https://code.google.com/p/android/issues/detail?id=51195 (accessed 30 September 2013).

20. Android Open Source Project - Issue Tracker, Issue #8407: HTC Desire Android 2.1-Update1 Does Not Receive UDP Broadcasts. http://code.google.com/p/android /issues/detail?id=8407 (accessed 30 September 2013).

21. Android Open Source Project - Issue Tracker. https://code.google.com/p/android/issues/list? can=2&q=multicast&colspec=ID+Type+Status+Owner+Summary+Stars&cells=tiles (accessed 30 September 2013).

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
2 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL IMAL HRA MAIL & RECORDS MGMT
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
3 (PDF)	DIR USARL RDRL CIH S D BRUNO D DORIA J FLETCHER